REMARKS/ARGUMENTS

Claims 1, 3, 7, 13 and 16 have been amended to further clarify the subject matter regarded as the invention. Claims 4, 5, 9, 19 and 20 have been cancelled. Thus, claims 1-3, 6-8 and 10-18 remain pending.

In the Office Action, the Examiner has also asserted that trademark language should be capitalized and accompanied by generic terminology (Office Action, item 1 on page 2). The Applicant has capitalized trademark names in an attempt to respect the proprietary nature of the marks and to make every effort to prevent their use in a manner that might adversely affect their validity as trademarks, pursuant to guidelines set forth in MPEP §608.01(v). It is respectfully submitted that the undersigned earnestly believes that there is no legal requirement that would suggest that <u>every</u> instance of trademark language must be accompanied by generic terminology. Furthermore, it is respectfully submitted that some programming languages or specific software products are <u>well known</u> to those skilled in the art by a trademark name (e.g., Java programming language). As such, it is NOT necessary to accompany trademark names such as the Java programming language with generic terminology. In fact, it may be inappropriate to use generic terminology if a programming language or product is <u>primarily</u> known to those skilled in the art by its trademark name (e.g., Java programming language). (Please see, for example, the Board of Appeals decision to the effect that a product on the market should be <u>known generally to those skilled in the art</u> and <u>necessity</u> for using a trade name; *Ex parte Frederick and Waterfall*, 75 USPQ 298 (Bd. Pat. App. & Int. 1947).) The Applicant has painstakingly made every effort to expedite prosecution of the present application and respectfully requests that the Examiner withdraw all rejections.

In the Office Action, the Examiner rejected claims 1-4, 13-14 and 16-19 under 35 U.S.C. §102(e) as being anticipated by U.S. Patent Application No. 6,412,108 (*Blandy et al.*). It is noted that *Blandy et al.* teaches reading bytecode instructions during Java Bytecode verification. *Blandy et al.,* however, teaches optimization of the method after the Bytecode verification process (i.e., after class loading and linking), but prior to initialization and execution of the method (*Blandy et al.*, Col. 4, lines 63-67). Accordingly, it is respectfully submitted that *Blandy et al.* does not teach or suggest counting, during Bytecode verification, the number of times a sequence of Java Bytecode instructions appears in a stream of Java Bytecode instructions. In addition, it is respectfully submitted that *Blandy et al.* does not teach or suggest determining during Bytecode verification whether said sequence of Java Bytecode instructions should be represented by one instruction, or generated during Bytecode verification. In fact, *Blandy et al.* teaches away from performing these operations during Bytecode verification because it teaches first loading and verifying the method before analyzing it for optimization (*Blandy et al.,* Col. 2, lines 40-44, Col. 4, lines 63-67). Accordingly, it is respectfully submitted that claim 6 and 13 and their dependent claims are patentable over *Blandy et al.* for at least these reasons.

In the Office Action, the Examiner has noted that *Blandy et al.* does not teach an internal representation of a pair of streams in the context of the claimed invention (Office Action, paragraph 4 on page 4). However, the Examiner has asserted that *Tommesani* teaches representing processing of a machine through an architecture known as SISD which teaches "a two stream representation of processing in the form of an instruction stream and a data stream" (Office Action, paragraph 4 on pages 4-5). It is noted that *Tommesani* describes that a SISD computer consists of a single processing unit receiving a single instruction stream that operates on a single stream of data. However, it should also be noted that *Tommesan*i illustrates a programming model invented by John von Neuman in the 1940's where, at each processing step, a control unit emits one instruction that operates on a datum (data) obtained from a memory unit (*Tommesan*i, text and accompanying figure on page 1). However, it is very respectfully submitted that the SISD computer does not teach generating an internal representation of a Java macro instruction in a pair of streams in a Java virtual machine. It should be noted that the pair of streams collectively represent an internal representation for an input stream of Java Bytecode instructions in said Java virtual machine. Accordingly, it is respectfully submitted that claims 1 and 16 and their dependent claims are patentable over *Blandy et al.* and *Tommesan*i taken alone, or in any proper combination for these reasons alone. Moreover, these dependent claims recite additional features which render them patentable for additional reasons.

Based on the foregoing, it is submitted that all pending claims are patentably distinct over the cited art of record. Additional limitations recited in the independent claims or the dependent claims are not further discussed because the limitations discussed above are sufficient to distinguish the claimed invention from the cited art. Accordingly, Applicant believes that all pending claims are allowable and respectfully requests a Notice of Allowance for this application from the Examiner.

Applicants hereby petition for an extension of time which may be required to maintain the pendency of this case, and any required fee for such extension or any further fee required in connection with the filing of this Amendment is to be charged to Deposit Account No. 500388 (Order No. SUN1P828). Should the Examiner believe that a telephone conference would expedite the prosecution of this application, the undersigned can be reached at the telephone number set out below.

Respectfully submitted,
BEYER WEAVER & THOMAS, LLP

[illegible]

R. Mahboubian
Reg. No. 44,890

P.O. Box 778
Berkeley, CA 94704-0778
(650) 961-8300

Atty. Docket No.: SUN1P828/P6116

#6/SL
Spe
4-1-04

SUBSTITUTE SPECIFICATION

WITH REDLINING

Attorney Docket No. SUN1P828/P6116

PATENT APPLICATION

FRAMEWORKS FOR GENERATION OF JAVA™ MACRO INSTRUCTIONS IN JAVA™ COMPUTING ENVIRONMENTS

Inventors:

Stepan Sokolov
34832 Dorado Common
Fremont, CA 94555
Citizenship: Ukraine

David Wallman
777 S. Mathilda Ave., #266
Sunnyvale, CA 94087
Citizenship: Israel

Assignee:

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303

BEYER WEAVER & THOMAS, LLP
P.O. Box 778
Berkeley, CA 94704-0778
Telephone (650) 961-8300

FRAMEWORKS FOR GENERATION OF JAVA™ MACRO INSTRUCTIONS IN JAVA™ COMPUTING ENVIRONMENTS

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application is related to concurrently filed U.S. Patent Application No. 09/939,310 entitled "FRAMEWORKS FOR GENERATION OF JAVA MACRO INSTRUCTIONS FOR INSTANTIATING JAVA OBJECTS," (Atty. Docket No. SUN1P839/P6719) which is hereby incorporated herein by reference for all purposes.

[0002] This application is related to concurrently filed U.S. Patent Application No. 09/938,915 entitled "FRAMEWORKS FOR GENERATION OF JAVA MACRO INSTRUCTIONS FOR PERFORMING PROGRAMMING LOOPS," (Atty. Docket No. SUN1P840/P6721) which is hereby incorporated herein by reference for all purposes.

[0003] This application is related to concurrently filed U.S. Patent Application No. 09/939,106 entitled "FRAMEWORKS FOR GENERATION OF JAVA MACRO INSTRUCTIONS FOR STORING VALUES INTO LOCAL VARIABLES," (Atty. Docket No. SUN1P842/P6723) which is hereby incorporated herein by reference for all purposes.

[0004] This application is related to U.S. Patent Application No. 09/819,120, filed March 27, 2001 (Att.Dkt.No. SUN1P811/P5512), entitled "REDUCED INSTRUCTION SET FOR JAVA VIRTUAL MACHINES," and hereby incorporated herein by reference for all purposes.

[0005] This application is related to U.S. Patent Application No. 09/703,449, filed October 31, 2000 (Att.Dkt.No. SUN1P814/P5417), entitled "IMPROVED FRAMEWORKS FOR LOADING AND EXECUTION OF OBJECT-BASED PROGRAMS," which is hereby incorporated herein by reference for all purposes.

[0006] This application is related to U.S. Patent Application No. 09/820,097, filed March 27, 2001 (Att.Dkt.No. SUN1P827/P6095), entitled

"ENHANCED VIRTUAL MACHINE INSTRUCTIONS," which is also hereby incorporated herein by reference for all purposes.

BACKGROUND OF THE INVENTION

[0007] The present invention relates generally to Java™ programming environments, and more particularly, to frameworks for generation of Java™ macro instructions in Java™ computing environments.

[0008] One of the goals of high level languages is to provide a portable programming environment such that the computer programs may easily be ported to another computer platform. High level languages such as "C" provide a level of abstraction from the underlying computer architecture and their success is well evidenced from the fact that most computer applications are now written in a high level language.

[0009] Portability has been taken to new heights with the advent of the World Wide Web ("the Web") which is an interface protocol for the Internet that allows communication between diverse computer platforms through a graphical interface. Computers communicating over the Web are able to download and execute small applications called applets. Given that applets may be executed on a diverse assortment of computer platforms, the applets are typically executed by a Java™ virtual machine.

[0010] Recently, the Java™ programming environment has become quite popular. The Java™ programming language is a language that is designed to be portable enough to be executed on a wide range of computers ranging from small devices (e.g., pagers, cell phones and smart cards) up to supercomputers. Computer programs written in the Java™ programming language (and other languages) may be compiled into Java™ Bytecode instructions that are suitable for execution by a Java™ virtual machine implementation. The Java™ virtual machine is commonly implemented in software by means of an interpreter for the Java™ virtual machine instruction set but, in general, may be software, hardware, or both. A particular Java™ virtual machine implementation and corresponding support libraries together constitute a Java™ runtime environment.

[0011] Computer programs in the Java™ programming language are arranged in one or more classes or interfaces (referred to herein jointly as classes or class files). Such programs are generally platform, i.e., hardware and operating system, independent. As such, these computer programs may be executed, without modification, on any computer that is able to run an implementation of the Java™ runtime environment.

[0012] Object-oriented classes written in the Java™ programming language are compiled to a particular binary format called the "class file format." The class file includes various components associated with a single class. These components can be, for example, methods and/or interfaces associated with the class. In addition, the class file format can include a significant amount of ancillary information that is associated with the class. The class file format (as well as the general operation of the Java™ virtual machine) is described in some detail in The Java Virtual Machine Specification, Second Edition, by Tim Lindholm and Frank Yellin, which is hereby incorporated herein by reference.

[0013] Fig. 1A shows a progression of a simple piece of a Java™ source code 101 through execution by an interpreter, the Java™ virtual machine. The Java™ source code 101 includes the classic Hello World program written in Java™. The source code is then input into a Bytecode compiler 103 that compiles the source code into Bytecodes. The Bytecodes are virtual machine instructions as they will be executed by a software emulated computer. Typically, virtual machine instructions are generic (*i.e.*, not designed for any specific microprocessor or computer architecture) but this is not required. The Bytecode compiler outputs a Java™ class file 105 that includes the Bytecodes for the Java™ program. The Java™ class file is input into a Java™ virtual machine 107. The Java™ virtual machine is an interpreter that decodes and executes the Bytecodes in the Java™ class file. The Java™ virtual machine is an interpreter, but is commonly referred to as a virtual machine as it emulates a microprocessor or computer architecture in software (e.g., the microprocessor or computer architecture may not exist in hardware).

[0014] Fig. 1B illustrates a simplified class file 100. As shown in Fig. 1B, the class file 100 includes a constant pool 102 portion, interfaces portion 104, fields portion 106, methods portion 108, and attributes portion 110. The methods portion 108 can include, or have references to, several Java™ methods associated with the Java™ class which is represented in the class file 100. One of these methods is an initialization method used to initialize the Java™ class after the class file has been loaded by the virtual machine but before other methods can be invoked. In other words, typically, an initialization method is used to initialize a Java™ class before the classes can be used.

[0015] A conventional virtual machine's interpreter decodes and executes the Java™ Bytecode instructions, one instruction at a time, during execution, e.g., "at runtime." Typically, several operations have to be performed to obtain the information that is necessary to execute a Java™ instruction. Furthermore, there is a significant overhead associated with dispatching Bytecode instructions. In other words, the Java™ interpreter has to perform a significant amount of processing in order to switch from one instruction to the next. Accordingly, it is highly desirable to reduce the number of times the interpreter has to dispatch instructions. This, in turn, can improve the performance of virtual machines, especially those operating with limited resources.

[0016] In view of the foregoing, improved frameworks for execution of Java™ Bytecode instructions are needed.

SUMMARY OF THE INVENTION

[0017] Broadly speaking, the invention relates to Java™ programming environments, and more particularly, to frameworks for generation of Java™ macro instructions in Java™ computing environments. Accordingly, techniques for generation of Java™ macro instructions suitable for use in Java™ computing environments are disclosed. As such, the techniques can be implemented in a Java™ virtual machine to efficiently execute Java™ instructions. As will be appreciated, a Java™ macro instruction can be substituted for two or more Java™ Bytecode instructions. This, in turn, reduces the number of Java™ instructions that are executed by the interpreter. As a result, the performance of virtual machines, especially those operating with limited resources, is improved.

[0018] The invention can be implemented in numerous ways, including as a method, an apparatus, a computer readable medium, and a database system. Several embodiments of the invention are discussed below.

[0019] As a method for generating a Java™ macro instruction corresponding to one or more Java™ Bytecode instructions, one embodiment of the invention includes the acts of: reading a stream of Java™ Bytecode instructions; determining whether two or more Java™ Bytecode instructions in the Java™ Bytecode stream can be represented by one instruction; generating a Java™ macro instruction that represents the two or more Java™ Bytecode instructions when two or more Java™ Bytecode instructions in the Java™ Bytecode stream can be represented by one instruction. The Java™ macro instruction is suitable for execution by a Java™ virtual machine, and when executed, the Java™ macro instruction can operate to perform one or more operations that are performed by the two or more Java™ Bytecode instructions.

[0020] As a method of generating a Java™ macro instruction corresponding to one or more Java™ Bytecode instructions, one embodiment of the invention includes the acts of: reading a stream of Java™ Bytecode instructions; counting the number of times a sequence of Java™ Bytecode instructions appears in the stream of Java™ Bytecode instructions, the sequence of Java™ Bytecode instructions including two or more Java™ Bytecode instructions which are in a sequence in the stream; determining whether the sequence of Java™ Bytecode instructions should be represented by one instruction; generating a Java™ macro instruction that represents the sequence of Java™ Bytecode instructions when the sequence of Java™ Bytecode instructions can be represented by the one instruction. The Java™ macro instruction is suitable for execution by a Java™ virtual machine. When executed, the Java™ macro instruction can operate to perform one or more operations that are performed by the sequence of Java™ Bytecode instructions.

[0021] Another embodiment of the invention provides a Java™ macro instruction generator suitable for generation of Java™ macro instructions, wherein each Java™ macro instruction corresponds to one or more Java™ Bytecode instructions. The Java™ macro instruction generator operates to: read a stream of Java™ Bytecode instructions during Java™ Bytecode verification; determine whether two or more Java™ Bytecode instructions in the Java™ Bytecode stream can be represented by one instruction; generate a Java™ macro instruction that represents the two or more Java™ Bytecode instructions when two or more Java™ Bytecode instructions in the Java™ Bytecode stream can be represented by one instruction. The Java™ macro instruction is suitable for execution by a Java™ virtual machine, and when executed, the Java™ macro instruction can operate to perform one or more operations that are performed by the two or more Java™ Bytecode instructions.

[0022] These and other aspects and advantages of the present invention will become more apparent when the detailed description below is read in conjunction with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

[0023] The present invention will be readily understood by the following detailed description in conjunction with the accompanying drawings, wherein like reference numerals designate like structural elements, and in which:

Fig. 1A shows a progression of a simple piece of a Java™ source code through execution by an interpreter, the Java™ virtual machine.

Fig. 1B illustrates a simplified class file.

Figs. 2A-2B illustrate Java™ computing environments including Java™ macro instruction generators.

Fig. 3 illustrates a method for generating Java™ macro instructions in accordance with one embodiment of the invention.

Fig. 4 illustrates a method for generating Java™ macro instructions in accordance with another embodiment of the invention.

Fig. 5 illustrates a Java™ Bytecode verifier in accordance with one embodiment of the invention.

Figs. 6A-6B illustrate Java™ computing environments including Java™ macro instruction generators and Java™ Bytecode translators in accordance with one embodiment of the invention.

Fig. 7A illustrates a computing environment including an internal representation of an inventive "DUP" instruction suitable for duplicating values on the stack in accordance with one embodiment of the invention.

Figs. 7B-7C illustrate some of the Java™ Bytecode instructions described in Fig. 7A.

Fig. 8 illustrates a mapping of Java™ Bytecode instantiation instructions to the virtual machine instructions provided in accordance with one embodiment of the invention.

Fig. 9A illustrates another sequence of conventional Java™ Bytecodes that can be executed frequently by a Java™ interpreter.

Fig. 9B illustrates a Java™ computing environment including a Java™ macro instruction generator and a Java™ Bytecode translator in accordance with another embodiment of the invention.

Fig. 10A illustrates an internal representation of a set of Java™ "Load" instructions suitable for loading values from a local variable in accordance with another embodiment of the invention.

Fig. 10B illustrates a set of Java™ Bytecode instructions for loading 4 byte local variables that can be represented by an inventive "Load" command in accordance with one embodiment of the invention.

Fig. 10C illustrates a set of Java™ Bytecode instructions for loading 8 byte local variables in accordance with one embodiment of the invention.

Figs. 11A and 11B illustrate some Java™ conventional Bytecode instructions for performing conditional flow operations which can be represented by two inventive virtual machine instructions in accordance with one embodiment of the invention.

Fig. 12A illustrates yet another sequence of conventional Java™ Bytecodes that can be executed frequently by a Java™ interpreter.

Fig. 12B illustrates the Java™ Bytecode translator operating to translate conventional Java™ instructions into inventive Java™ instructions.

Fig. 13A illustrates a computing environment in accordance with one embodiment of the invention.

Figs. 13B and 13C illustrate a set of conventional Java™ Bytecode instructions for storing arrays that can be represented by an inventive virtual machine instruction (e.g., Astore) in accordance with one embodiment of the invention.

DETAILED DESCRIPTION OF THE INVENTION

[0024] As described in the background section, the Java™ programming environment has enjoyed widespread success. Therefore, there are continuing efforts to extend the breadth of Java™ compatible devices and to improve the performance of such devices. One of the most significant factors influencing the performance of Java™ based programs on a particular platform is the performance of the underlying virtual machine. Accordingly, there have been extensive efforts by a number of entities to improve performance in Java™ compliant virtual machines.

[0025] To achieve this and other objects of the invention, techniques for generation of Java™ macro instructions suitable for use in Java™ computing environments are disclosed. As such, the techniques can be implemented in a Java™ virtual machine to efficiently execute Java™ instructions. As will be appreciated, a Java™ macro instruction can be substituted for two or more Java™ Bytecode instructions. This, in turn, reduces the number of Java™ instructions that are executed by the interpreter. As a result, the performance of virtual machines, especially those operating with limited resources, is improved.

[0026] Embodiments of the invention are discussed below with reference to Figs. 2A-13C. However, those skilled in the art will readily appreciate that the detailed description given herein with respect to these figures is for explanatory purposes only as the invention extends beyond these limited embodiments.

[0027] Fig. 2A illustrates a Java™ computing environment 200 in accordance with one embodiment of the invention. The Java™ computing environment 200 includes a Java™ macro instruction generator 202 suitable for generation of macro instructions which are suitable for execution by an interpreter. As shown in Fig. 2A, the Java™ macro instruction generator 202 can read a stream of Java™ Bytecode instructions 204 (Java™ Bytecode instructions 1-N). Moreover, the Java™

macro instruction generator 202 can produce a Java™ macro instruction 206 which represents two or more Java™ Bytecode instructions in the stream 204.

[0028] The Java™ Bytecode instructions in the stream 204 can be conventional Java™ Bytecode instructions, for example, conventional instructions "new" and "dup" which typically appear in sequence in order to instantiate a Java™ object. As will be appreciated by those skilled in the art, certain sequences appear frequently during the execution of Java™ programs. Thus, replacing such sequences with a single macro instruction can reduce the overhead associated with dispatching Java™ Bytecode instructions. As a result, the performance of virtual machines, especially those operating with limited resources, is enhanced.

[0029] It should be noted that the Java™ macro instruction generator 202 can also be used in conjunction with a Java™ Bytecode translator in accordance with one preferred embodiment of the invention. Referring now to Fig. 2B, a Java™ Bytecode translator 230 operates to translate conventional Java™ instructions 1-M into inventive Java™ instructions 234 (1-N), wherein N is an integer less than the integer M. More details about the Java™ Bytecode translator 230 and inventive Java™ instructions 1-N are described in U.S. Patent Application No. 09/819,120 (Att.Dkt.No. SUN1P811/P5512), entitled "REDUCED INSTRUCTION SET FOR JAVA VIRTUAL MACHINES," and U.S. Patent Application No. 09/820,097 (Att.Dkt.No. SUN1P827/P6095), entitled "ENHANCED VIRTUAL MACHINE INSTRUCTIONS." As will be appreciated, the use of the inventive Java™ instructions in conjunction with the Java™ macro instruction generator can further enhance the performance of virtual machines.

[0030] It should also be noted that the Java™ macro instruction can be internally represented in the virtual machine as a pair of Java™ streams in accordance with one embodiment of the invention. The pair of Java™ streams can be a code stream and a data stream. The code stream is suitable for containing the code portion of Java™ macro instructions, and the data stream is suitable for containing a data portion of said Java™ macro instruction. More details about representing instructions as a pair of streams can be found in the U.S. Patent Application No. 09/703,449 (Att.Dkt.No. SUN1P814/P5417), entitled "IMPROVED FRAMEWORKS FOR LOADING AND EXECUTION OF OBJECT-BASED PROGRAMS."

[0031] Fig. 3 illustrates a method 300 for generating Java™ macro instructions in accordance with one embodiment of the invention. The method 300 can be used, for example, by the Java™ macro instruction generator 202 of Figs. 2A-B. Initially, at operation 302, a stream of Java™ Bytecode instructions are read. As will be appreciated, the stream of Java™ Bytecode instructions can be read during the Bytecode verification phase. Java™ Bytecode verification is typically performed in order to ensure the accuracy of Java™ instructions. As such, operation 302 can be efficiently performed during Bytecode verification since typically there is a need to verify Bytecode instructions.

[0032] Next, at operation 304, a determination is made as to whether a predetermined sequence of two or more Java™ Bytecode instructions has been found. If it is determined at operation 304 that a predetermined sequence of two or more Java™ Bytecode instructions has not been found, the method 300 ends. However, if it is determined at operation 304 that a predetermined sequence of two or more Java™ Bytecode instructions has been found, the method 300 proceeds to operation 306 where a Java™ macro instruction that represents the sequence of two or more Java™ Bytecode instructions is generated. The method 300 ends following operation 306. It should be noted that operations 304 and 306 can also be performed during the Java™ Bytecode verification phase.

[0033] Fig. 4 illustrates a method 400 for generating Java™ macro instructions in accordance with another embodiment of the invention. The method 400 can be used, for example, by the Java™ macro instruction generator 202 of Figs. 2A-B. Initially, at operation 402, a stream of Java™ Bytecode instructions is read. Again, operation 402 can efficiently be performed during Bytecode verification since Bytecode verification is typically performed anyway.

[0034] Next, at operation 404, the number of times a sequence of Java™ Bytecode instructions appear in the stream of Java™ Bytecode instructions is counted. Thereafter, at operation 406, a determination is made as to whether the sequence has been counted for at least a predetermined number of times. If it is determined at operation 406 that the sequence has not been counted for at least a predetermined number of times, the method 400 ends. However, if it is determined at operation 406 that the sequence has been counted for at least a predetermined number of times, the method 400 proceeds to operation 408 where a Java™ macro instruction that represents the sequence of Java™ Bytecode instructions is generated. The method 400 ends following operation 408.

[0035] Fig. 5 illustrates a Java™ Bytecode verifier 500 in accordance with one embodiment of the invention. The Java™ Bytecode verifier 500 includes a sequence analyzer 502 suitable for analyzing a stream of Java™ Bytecodes 504. As shown in Fig. 5, the stream of Java™ Bytecodes 504 consists of a sequence of Java™ Bytecode instructions 1–N. The Java™ Bytecode verifier 500 operates to determine whether a sequence of two or more Java™ Bytecode instructions can be represented as a Java™ macro instruction. If the Bytecode verifier 500 determines that a sequence of two or more Java™ Bytecode instructions can be represented as a Java™ macro instruction, the Bytecode verifier 500 produces a Java™ macro instruction. The Java™ macro instruction corresponds to the sequence of two or more Java™ Bytecode instructions. Accordingly, the Java™ macro instruction can replace the sequence of two or more Java™ Bytecode instructions in the Java™ stream.

[0036] Referring to Fig. 5, a sequence of two or more Java™ Bytecode instructions 506 in the stream 504 can be identified by the Java™ Bytecode verifier 500. The sequence of two or more Java™ Bytecode instructions 506 (instructions I1-IM) can be located in positions K through (K+M-1) in the stream 504. After identifying the sequence of two or more Java™

Bytecode instructions 506, the Java™ Bytecode verifier 500 can operate to replace the sequence with a Java™ macro instruction 508 (I1-IM). As a result, the stream 504 is reduced to a stream 510 consisting of (N-M) Java™ Bytecode instructions. As will be appreciated, the Java™ Bytecode verifier 500 can identify a number of predetermined sequences of Java™ Bytecode instructions and replace them with the appropriate Java™ macro instruction. The Java™ Bytecode verifier 500 can also be implemented to analyze the sequences that appear in the stream 504 and replace only those that meet a criteria (e.g., a sequence that has appeared more than a predetermined number of times). In any case, the number of Java™ Bytecode instructions in an input stream 504 (e.g., stream 504) can be reduced significantly. Thus, the performance of virtual machines, especially those operating with limited resources, can be enhanced.

[0037] As noted above, the Java™ Bytecode instructions which are replaced in the stream can be conventional Java™ Bytecode instructions which often appear in a sequence. One such example is the various combinations of the conventional instructions representing "New$_x$" and "Dup$_x$" which typically appear in sequence in order to instantiate a Java™ object (e.g., New-Dup, Newarray-Dup_x1, Anewarray-Dup_x2 , etc.).

[0038] Fig. 6A illustrates a Java™ computing environment 600 including a Java™ macro instruction generator 602 in accordance with one embodiment of the invention. Referring now to Fig. 6A, conventional Java™ Bytecode instructions "New$_x$" and "Dup$_x$" are depicted in a sequence 610. The sequence 610 can be replaced by a single Java™ macro instruction "New-Dup" 612 by the Java™ macro instruction generator 602. As will be appreciated by those skilled in the art, the sequence 610 can appear frequently during the execution of Java™ programs. Thus, replacing this sequence with a single macro instruction can reduce the overhead associated with dispatching Java™ Bytecode instructions.

[0039] Again, it should be noted that the Java™ macro instruction 602 can also be used in conjunction with a Java™ Bytecode translator in accordance with one preferred embodiment of the invention. More details about the Java™ Bytecode translator and inventive Java™ Bytecode instructions are described in U.S. Patent Application No. 09/819,120 (Att.Dkt.No. SUN1P811/P5512), entitled "REDUCED INSTRUCTION SET FOR JAVA VIRTUAL MACHINES," and U.S. Patent Application No. 09/820,097 (Att.Dkt.No. SUN1P827/P6095), entitled "ENHANCED VIRTUAL MACHINE INSTRUCTIONS."

[0040] Fig. 6B illustrates a Java™ computing environment 620, including a Java™ macro instruction generator 602 and a Java™ Bytecode translator 622, in accordance with one embodiment of the invention. Referring now to Fig. 6B, the Java™ Bytecode translator 622 operates to translate conventional Java™ instructions 610 into inventive Java™ instructions 630. The Java™ macro instruction generator 602 can receive the inventive Java™ instructions 630 and generate a corresponding Java™ macro instruction "New-Dup" 624.

[0041] It should be noted that the inventive Java™ instructions 630 represent a reduced set of Java™ instructions suitable for execution by a Java™ virtual machine. This means that the number of instructions in the inventive reduced set is significantly less than the number of instructions in the conventional Java™ Bytecode instruction set. Furthermore, the inventive Java™ instructions provide for inventive operations that cannot be performed by conventional Java™ Bytecode instructions. By way of example, an inventive virtual machine operation "DUP" (shown in sequence 630) can be provided in accordance with one embodiment of the invention. The inventive virtual machine instruction DUP allows values in various positions on the execution stack to be duplicated on the top of the execution stack.

[0042] Fig. 7A illustrates a computing environment 700 including an internal representation 701 of an inventive "DUP" instruction 702 suitable for duplicating values on the stack in accordance with one embodiment of the invention. The internal representation 701 includes a pair of streams, namely, a code stream 706 and a data stream 708. In the described embodiment, each entry in the code stream 706 and data stream 708 represents one byte. The inventive virtual machine instruction DUP 702 is associated with a data parameter A in the code stream 706. It should be noted that data parameter A may also be implemented in the data stream 708. In any case, the data parameter A indicates which 4 byte value (word value) on an execution stack 704 should be duplicated on the top of the execution stack 704. The data parameter A can indicate, for example, an offset from the top of the execution stack 704. As shown in Fig. 7A, the data parameter A can be a reference to "Wi," a word (4 byte) value on the execution stack. Accordingly, at execution time, the virtual machine can execute the "DUP" command 702. As a result, the Wi word will be duplicated on the top of the stack. Thus, the inventive "DUP" instruction can effectively replace various Java™ Bytecode instructions that operate to duplicate 4 byte values on top of the execution stack. Fig. 7B illustrates some of these Java™ Bytecode instructions. Similarly, as illustrated in Fig. 7C, an inventive "DUPL" instruction can be provided to effectively replace various Java™ Bytecode instructions that operate to duplicate 8 byte values (2 words) on top of the execution stack.

[0043] It should be noted that conventional Java™ Bytecode "Dup$_x$" instructions only allow for duplication of values in certain positions on the execution stack (i.e., conventional instructions Dup, Dup_x1 and Dup_x2 respectively allow duplication of the first, second and third words on the execution stack). However, the inventive instructions "DUP" and "DUPL" can be used to duplicate a much wider range of values on the execution stack (e.g., W4, Wi, WN, etc.).

[0044] Referring back to Fig. 6B, another inventive instruction, Java™ Bytecode instruction "New" is shown in the sequence 630. The Java™ Bytecode instruction "New" can effectively replace various conventional Java™ Bytecodes used for instantiation.

[0045] Fig. 8 illustrates a mapping of Java™ Bytecode instantiation instructions to the virtual machine instructions provided in accordance with one embodiment of the invention. As will be appreciated, the four conventional Java™ Bytecode instructions can effectively be mapped into a single virtual machine instruction (e.g., NEW). The virtual machine instruction NEW operates to instantiate objects and arrays of various types. In one embodiment, the inventive virtual machine instruction NEW operates to determine the types of the objects or arrays based on the parameter value of the Java™ Bytecode instantiation instruction. As will be appreciated, the Java™ Bytecode instructions for instantiation are typically followed by a parameter value that indicates the type. Thus, the parameter value is readily available and can be used to allow the NEW virtual machine instruction to instantiate the appropriate type at execution time.

[0046] Fig. 9A illustrates another sequence 902 of conventional Java™ Bytecodes that can be executed frequently by a Java™ interpreter. The sequence 902 represents an exemplary sequence of instructions that are used in programming loops. As such, sequences, such as the sequence 902, can be repeated over and over again during the execution of Java™ Bytecode instructions. As shown in Fig. 9A, the Java™ macro instruction generator 202 can replace the conventional sequence of Java™ instructions "iinc," "iload," and "if_cmplt" with a Java™ macro instruction "Loop1."

[0047] Fig. 9B illustrates a Java™ computing environment 900, including a Java™ macro instruction generator 902 and a Java™ Bytecode translator 904, in accordance with one embodiment of the invention. Referring now to Fig. 9B, the Java™ Bytecode translator 904 operates to translate conventional Java™ instructions 910 into inventive Java™ instructions 920. The Java™ macro instruction generator 902 can receive the inventive Java™ instructions 920 and generate a corresponding Java™ macro instruction "Loop1" 940.

[0048] One of the inventive instructions in the sequence 920 is the inventive instruction "Load." Fig. 10A illustrates an internal representation 1000 of a set of Java™ "Load" instructions suitable for loading values from a local variable in accordance with another embodiment of the invention. In the described embodiment, a code stream 1002 of the internal representation 1000 includes a Load command 1006 representing an inventive virtual machine instruction suitable for representation of one or more Java™ "Load from a local variable" Bytecode instructions. It should be noted that the Load command 1006 has a one byte parameter associated with it, namely, an index i 1008 in the data stream 1004. As will be appreciated, at run time, the Load command 1006 can be executed by a virtual machine to load (or push) a local variable on top of the execution stack 1020. By way of example, an offset 0 1022 can indicate the starting offset for the local variables stored on the execution stack 1020. Accordingly, an offset i 1024 identifies the position in the execution stack 1020 which corresponds to the index i 1008.

[0049] It should be noted that in the described embodiment, the Load command 1006 is used to load local variables as 4 bytes (one word). As a result, the value indicated by the 4 bytes A, B, C and D (starting at offset i 1024) is loaded on the top of the execution stack 1020 when the Load command 1006 is executed. In this manner, the Load command 1006 and index i 1008 can be used to load (or push) 4 byte local variables on top of the execution stack at run time. As will be appreciated, the Load command 1006 can effectively represent various conventional Java™ Bytecode instructions. Fig. 10B illustrates a set of Java™ Bytecode instructions for loading 4 byte local variables that can be represented by an inventive "Load" command in accordance with one embodiment of the invention.

[0050] It should be noted that the invention also provides for loading local variables that do not have values represented by 4 bytes. For example, Fig. 10C illustrates a set of Java™ Bytecode instructions for loading 8 byte local variables in accordance with one embodiment of the invention. As will be appreciated, all of the Java™ Bytecode instructions listed in Fig. 10C can be represented by a single inventive virtual machine instruction (e.g., a "LoadL" command). The "LoadL" command can operate, for example, in a similar manner as discussed above.

[0051] Referring back to Fig. 9B, the Java$^{\underline{TM}}$ Bytecode translator 904 operates to replace the conventional Bytecode instruction "if_cmplt" in the sequence 910 with the two Bytecode instructions "OP_ISUB" and "OP_JMPLT" in the reduced set of Java$^{\underline{TM}}$ Bytecode instructions. As will be appreciated, two or more of the inventive virtual machine instructions can be combined to perform relatively more complicated operations in accordance with one embodiment of the invention. By way of example, the conditional flow control operation performed by the Java$^{\underline{TM}}$ Bytecode instruction "lcmp" (compare two long values on the stack and, based on the comparison, push 0 or 1 on the stack) can effectively be performed by performing an inventive virtual machine instruction LSUB (Long subdivision) followed by another inventive virtual machine instruction JMPEQ (Jump if equal). Figs. 11A and 11B illustrate some conventional Java$^{\underline{TM}}$ Bytecode instructions for performing conditional flow operations which can be represented by two inventive virtual machine instructions in accordance with one embodiment of the invention.

[0052] Fig. 12A illustrates yet another sequence 1210 of conventional Java$^{\underline{TM}}$ Bytecodes that can be executed frequently by a Java$^{\underline{TM}}$ interpreter. The sequence 1210 represents an exemplary sequence of instructions that operate to obtain a field value and put it on the execution stack. As shown in Fig. 12A, the Java$^{\underline{TM}}$ macro instruction generator 602 can replace the conventional sequence 1210 of Java$^{\underline{TM}}$ instructions "Getfield" and "Astore$_x$" with a Java$^{\underline{TM}}$ macro instruction "Get_Store" 1212. The conventional instruction "Astore$_x$" represents various conventional Java$^{\underline{TM}}$ instructions used to store values on the execution stack.

[0053] Fig. 12B illustrates a Java$^{\underline{TM}}$ computing environment 1200, including a Java$^{\underline{TM}}$ macro instruction generator 602 and a Java$^{\underline{TM}}$ Bytecode translator 622, in accordance with one embodiment of the invention. Referring now to Fig. 12B, the Java$^{\underline{TM}}$ Bytecode translator 622 operates to translate conventional Java$^{\underline{TM}}$ instructions 1210 into inventive Java$^{\underline{TM}}$ instructions 1220. The Java$^{\underline{TM}}$ macro instruction generator 602 can receive the inventive Java™ instructions 1220 and generate a corresponding Java™ macro instruction "Resolve_Astore" 1222.

[0054] The inventive instruction "Astore" represents a virtual machine instruction suitable for storing values into arrays. By way of example, Fig. 13A illustrates a computing environment 1320 in accordance with one embodiment of the invention. An inventive AStore 1322 (store into array) virtual machine instruction can be used to store various values from the execution stack 1304 into different types of arrays in accordance with one embodiment of the invention. Again, the header 1310 of the array 1302 can be read to determine the array's type. Based on the array's type, the appropriate value (i.e., the appropriate number of bytes N on the execution stack 1304) can be determined. This value can then be stored in the array 1302 by using the array-index 1326. Thus, the inventive virtual machine instruction AStore can effectively represent various Java™ Bytecode instructions that are used to store values into an array. Figs. 13B and 13C illustrate a set of conventional Java™ Bytecode instructions for storing arrays that can be represented by an inventive virtual machine instruction (e.g., Astore) in accordance with one embodiment of the invention.

[0055] Appendix A illustrates mapping of a set of conventional Java™ Bytecode instructions to one or more of the inventive virtual machine instructions listed in the right column.

[0056] The many features and advantages of the present invention are apparent from the written description, and thus, it is intended by the appended claims to cover all such features and advantages of the invention. Further, since numerous modifications and changes will readily occur to those skilled in the art, it is not desired to limit the invention to the exact construction and operation as illustrated and described. Hence, all suitable modifications and equivalents may be resorted to as falling within the scope of the invention.

What is claimed is:

CLAIMS

1. A method of generating a Java macro instruction corresponding to one or more Java Bytecode instructions, said method comprising:

reading a stream of Java Bytecode instructions;

determining whether two or more Java Bytecode instructions in said Java Bytecode stream can be represented by one instruction;

generating a Java macro instruction that represents said two or more Java Bytecode instructions when said determining determines that two or more Java Bytecode instructions in said Java Bytecode stream can be represented by one instruction;

wherein said Java macro instruction is suitable for execution by a Java virtual machine; and

wherein, when executed, said Java macro instruction can operate to perform one or more operations that are performed by said two or more Java Bytecode instructions.

2. A method as recited in claim 1,

wherein said determining operates to determine whether a predetermined sequence of two or more Java Bytecode instructions have been found.

3. A method as recited in claim 1, wherein said reading is performed during Java Bytecode verification by said virtual machine.

4. A method as recited in claim 1, wherein said method further comprises:

replacing said two or more Java Bytecode instructions with said Java macro instruction, and

wherein said macro instruction is internally represented in said virtual machine.

5. A method as recited in claim 4, wherein said internal representation comprises a pair of streams.

6. A method as recited in claim 5,

wherein said pair of streams includes a code stream and a data stream,

wherein said code stream is suitable for containing a code portion of said Java macro instruction, and

wherein said data stream is suitable for containing a data portion of said Java macro instruction.

7. A method of generating a Java macro instruction corresponding to one or more Java Bytecode instructions, said method comprising:

reading a stream of Java Bytecode instructions;

counting the number of times a sequence of Java Bytecode instructions appears in said stream of Java Bytecode instructions, said sequence of Java Bytecode instructions including two or more Java Bytecode instructions which are in a sequence in said stream;

determining whether said sequence of Java Bytecode instructions should be represented by one instruction;

generating a Java macro instruction that represents said sequence of Java Bytecode instructions when said determining determines that said sequence of Java Bytecode instructions can be represented by said one instruction;

wherein said Java macro instruction is suitable for execution by a Java virtual machine; and

wherein, when executed, said Java macro instruction can operate to perform one or more operations that are performed by said sequence of Java Bytecode instructions.

8. A method as recited in claim 7, wherein said determining of whether said sequence of Java Bytecode instructions should be represented by one

instruction operates to determine whether said sequence has been counted for at least a predetermined number of times.

9. A method as recited in claim 8, wherein said counting is performed during Bytecode verification.

10. A method as recited in claim 9, wherein said method further comprises:

replacing said two or more Java Bytecode instructions with said Java macro instruction, and

wherein said macro instruction is internally represented in said virtual machine.

11. A method as recited in claim 10, wherein said internal representation comprises a pair of streams.

12. A method as recited in claim 11,

wherein said pair of streams includes a code stream and a data stream,

wherein said code stream is suitable for containing a code portion of said Java macro instruction, and

wherein said data stream is suitable for containing a data portion of said Java macro instruction.

13. A method of generating a Java macro instruction corresponding to one or more Java Bytecode instructions, said method comprising:

reading a stream of Java Bytecode instructions during Java Bytecode verification;

determining whether two or more Java Bytecode instructions in said Java Bytecode stream can be represented by one instruction;

generating a Java macro instruction that represents said two or more Java Bytecode instructions when said determining determines that two or more Java Bytecode instructions in said Java Bytecode stream can be represented by one instruction;

wherein said Java macro instruction is suitable for execution by a Java virtual machine; and

wherein, when executed, said Java macro instruction can operate to perform one or more operations that are performed by said two or more Java Bytecode instructions.

14. A method as recited in claim 13,

wherein said determining operates to determine whether a predetermined sequence of two or more Java Bytecode instructions have been found.

15. A method as recited in claim 13,

wherein said method further comprises counting the number of times a sequence of Java Bytecode instructions appear in said stream, and

wherein said determining operates to determine whether a sequence has been counted for at least a predetermined number of times.

16. In a Java computing environment, a Java macro instruction generator suitable for generation of Java macro instructions,

wherein each Java macro instruction corresponds to one or more Java Bytecode instructions,

wherein said Java macro instruction generator operates to:

read a stream of Java Bytecode instructions during Java Bytecode verification;

determine whether two or more Java Bytecode instructions in said Java Bytecode stream can be represented by one instruction;

generate a Java macro instruction that represents said two or more Java Bytecode instructions when said determining determines that two or more Java Bytecode instructions in said Java Bytecode stream can be represented by one instruction;

wherein said Java macro instruction is suitable for execution by a Java virtual machine, and

wherein, when executed, said Java macro instruction can operate to perform one or more operations that are performed by said two or more Java Bytecode instructions.

17. A Java macro instruction generator as recited in claim 16, wherein said Java macro instruction generator operates during Java Bytecode verification.

18. A Java macro instruction generator as recited in claim 16, wherein said Java macro instruction generator operates to determine whether a predetermined sequence of two or more Java Bytecode instructions has been found.

19. A Java macro instruction generator as recited in claim 16,

operating to replace said two or more Java Bytecode instructions with said Java macro instruction, and

wherein said Java macro instruction is internally represented in said virtual machine.

20. A Java macro instruction generator as recited in claim 19, wherein said internal representation comprises a pair of streams.

FRAMEWORKS FOR GENERATION OF JAVA™ MACRO INSTRUCTIONS IN JAVA™ COMPUTING ENVIRONMENTS

ABSTRACT OF THE DISCLOSURE

[0057] Techniques for generation of Java™ macro instructions suitable for use in Java™ computing environments are disclosed. As such, the techniques can be implemented in a Java™ virtual machine to efficiently execute Java™ instructions. As will be appreciated, a Java™ macro instruction can be substituted for two or more Java™ Bytecode instructions. This, in turn, reduces the number of Java™ instructions that are executed by the interpreter. As a result, the performance of virtual machines, especially those operating with limited resources, is improved. A Java™ macro instruction can be generated for conventional Java™ instruction sequences or sequences of Java™ instruction that are provided in a reduced set of instruction. In any case, sequences that are frequently encountered can be replaced by a Java™ macro instruction. These sequences are typically encountered when Java™ objects are instantiated, during programming loops, and when a local variables are assigned a value.

Appendix A

nop	IGNORE_OPCODE
aconst_null	OP_PUSHB
iconst_ml	OP_PUSHB
iconst_0	OP_PUSHB
iconst_1	OP_PUSHB
iconst_2	OP_PUSHB
iconst_3	OP_PUSHB
iconst_4	OP_PUSHB
iconst_5	OP_PUSHB
lconst_0	OP_PUSHL
lconst_1	OP_PUSHL
fconst_0	OP_PUSH
fconst_1	OP_PUSH
fconst_2	OP_PUSH
dconst_0	OP_PUSHL
dconst_1	OP_PUSHL
bipush	OP_PUSHB
sipush	OP_PUSH
ldc	OP_PUSH
ldc_w	OP_PUSH
ldc2_w	OP_PUSHL
iload	OP_LOAD
lload	OP_LOADL
fload	OP_LOAD
d1oad	OP_LOADL
aload	OP_LOAD
iload_0	OP_LOAD
iload_1	OP_LOAD
iload_2	OP_LOAD
iload_3	OP_LOAD
lload_0	OP_LOADL
11oad_1	OP_LOADL
11oad_2	OP_LOADL

lload_3	OP_LOADL
fload_0	OP_LOADL
fload_1	OP_LOAD
fload_2	OP_LOAD
fload_3	OP_LOAD
dload_0	OP_LOADL
d1oad_1	OP_LOADL
dload_2	OP_LOADL
dload_3	OP_LOADL
aload_0	OP_LOAD
aload_I	OP_LOAD
aload_2	OP_LOAD
aload_3	OP_LOAD
iaload	OP_ALOAD
laload	OP_ALOAD
faload	OP_ALOAD
daload	OP_ALOAD
aaload	OP_ALOAD
baload	OP_ALOAD
caload	OP_ALOAD
saload	OP_ALOAD
istore	OP_STOR
lstore	OP_STORL
fstore	OP_STOR
dstore	OP_STORL
astore	OP_STOR
istore_0	OP_STOR
istore_1	OP_STOR
istore_2	OP_STOR
istore_3	OP_STOR
1store_0	OP_STORL
1store_1	OP_STORL
lstore_2	OP_STORL
1store_3	OP_STORL
fstore_0	OP_STOR
fstore_1	OP_STOR

fstore_2	OP_STOR
fstore_3	OP_STOR
dstore_0	OP_STORL
dstore_1	OP_STORL
dstore_2	OP_STORL
dstore_3	OP_STORL
astore_0	OP_STOR
astore_1	OP_STOR
astore_2	OP_STOR
astore_3	OP_STOR
iastore	OP_ASTORE
lastore	OP_ASTOREL
fastore	OP_ASTORE
dastore	OP_ASTOREL
aastore	OP_ASTORE
bastore	OP_ASTORE
castore	OP_ASTORE
sastore	OP_ASTORE
pop	OP_POP
pop2	OP_POP
dup	OP_DUP
dup_x1	OP_DUP
dup_x2	OP_DUP
dup2	OP_DUPL
dup2_x1	OP_DUPL
dup2_x2	OP_DUPL
swap	OP_SWAP
iadd	OP_IADD
ladd	OP_LADD
fadd	OP_FADD
dadd	OP_DADD
isub	OP_ISUB
lsub	OP_LSUB
fsub	OP_FSUB
dsub	OP_DSUB
imul	OP_IMUL

lmul	OP_LMUL
fmul	OP_FMUL
dmul	OP_DMUL
idiv	OP_IDIV
ldiv	OP_LDIV
fdiv	OP_FDIV
ddiv	OP_DDIV
irem	OP_IREM
lrem	OP_LREM
frem	OP_FREM
drem	OP_DREM
ineg	OP_INEG
lneg	OP_LNEG
fneg	OP_FNEG
dneg	OP_DNEG
ishl	OP_ISHL
lshl	OP_LSHL
ishr	OP_ISHR
lshr	OP_LSHR
iushr	OP_IUSHR
lushr	OP_LUSHR
iand	OP_IAND
land	OP_LAND
ior	OP_IOR
lor	OP_LOR
ixor	OP_IXOR
lxor	OP_LXOR
iinc	OP_IINC
i2l	OP_I2L
i2f	IGNORE_OPCODE
i2d	OP_I2D
12i	OP_L2I
12f	OP_L2F
12d	OP_L2D
f2i	IGNORE_OPCODE
f2l	OP_F2L

f2d	OP_F2D
d2i	OP_D2I
d2l	OP_D2L
d2f	OP_D2F
i2b	IGNORE_OPCODE
i2c	IGNORE_OPCODE
i2s	IGNORE_OPCODE
lcmp	OP_LSUB, OP_JMPEQ
fcmpl	OP_FSUB, OP_JMPLE
fcmpg	OP_FSUB, OP_JMPGE
dcmpl	OP_DCMP, OP_JMPLE
dcmpg	OP_DCMP, OP_JMPGE
ifeq	OP_JMPEQ
ifne	OP_JMPNE
iflt	OP_JMPLT
ifge	OP_JMPGE
ifgt	OP_JMPGT
ifle	OP_JMPLE
if icmpeq	OP_ISUB, OP_JMPEQ
if icmpne	OP_ISUB, OP_JMPNE
if icmplt	OP_ISUB, OP_JMPLT
if icmpge	OP_ISUB, OP_JMPGE
if icmpgt	OP_ISUB, OP_JMPGT
if icmple	OP_ISUB, OP_JMPLE
if acmpeq	OP_ISUB, OP_JMPEQ
if acmpne	OP_ISUB, OP_JMPNE
goto	OP_JMP
jsr	OP_JSR
ret	OP_RET
tableswitch	OP_SWITCH
lookupswitch	OP_SWITCH
ireturn	OP_RETURN
lreturn	OP_LRETURN
freturn	OP_RETURN
dreturn	OP_LRETURN
areturn	OP_RETURN

return	OP_RETURNV
getstatic	OP_RESOLVE
putstatic	OP_RESOLVEP
getfield	OP_RESOLVE
putfield	OP_RESOLVEP
invokevirtual	OP_RESOLVE
invokespecial	OP_RESOLVE
invokestatic	OP_RESOLVE
invokeinterface	OP_RESOLVE
xxxunusedxxx	IGNORE_OPCODE
new	OP_NEW
newarray	OP_NEW
anewarray	OP_NEW
arraylength	OP_ARRAYLENGTH
athrow	OP_THROW
checkcast	IGNORE_OPCODE
instanceof	OP_INSTANCEOF
monitorenter	OP_MUTEXINC
monitorexit	OP_MUTEXDEC
wide	OP_WIDE
multianewarray	OP_NEW
ifnull	OP_JMPEQ
ifnonnull	OP_JMPNE
goto_w	OP_JMP
jsr_w	OP_JSR